Automatic Detection of Logic Bugs in Hardware Designs

Alexander Klaiber, Sinclair Chau,
Transmeta Corporation
alex@klaiber.org, schau@transmeta.com

Abstract

The verification of complex microprocessor designs is a tough challenge; high-speed hardware emulators can help improve confidence in design correctness by greatly increasing verification throughput. Most attractively, their speed makes it realistic to simulate entire "real-life" application programs, hopefully extending test coverage. The usefulness of this approach is however greatly reduced by the fact that any bugs exposed by real application programs (as opposed to carefully constructed small test cases) are exceedingly difficult to isolate and identify.

We have developed a tool that eliminates this problem by completely automating the task of isolating bugs in the hardware design, by periodically comparing execution against a known-good reference model. This new approach has vastly increased the value of our emulation efforts, and found several bugs that escaped the normal test suite. Moreover, bugs that used to take weeks of painstaking work to isolate have been identified automatically by our tool in a matter of hours.

1. Introduction

Modern microprocessor designs have increased greatly in complexity; verifying the correctness of these designs has become an ever greater challenge. To increase confidence that the traditional testing has covered all the corner cases that arise in "real use", it is often desirable to simulate the execution of real-life applications on the new hardware design; a task that is made possible by the availability of high-speed hardware emulators, e.g., from Mentor or Quickturn.

This approach, however, has a major drawback that greatly limits its practical use: isolating and identifying a hardware bug that is only exposed by running a real application (or a long random test, for that matter) is very hard. This is because, unlike carefully crafted test cases, a real application is large, exercises many different functional blocks in the hardware, and is not self-checking. Furthermore, perturbing the system in order to isolate the cause of the failure — e.g., by recompiling the application (if that is even possible), inserting debugging breakpoints, or by changing system timing — may cause the bug to disappear[1]. Add to this the fact that despite their impressive speeds, hardware emulators are still orders of magnitude slower than real silicon, and debugging these failures becomes a very frustrating experience that can consume huge amounts of skilled manpower.

At Transmeta, for example, a previous-generation hardware emulation project uncovered a bug that prevented the booting of an operating system, and that did not show in any of the "traditional" test suites. Since Transmeta processors execute applications by dynamically translating them from x86 code to the processor's native code [Kla00], we were in the fortunate position that we could modify the application code indirectly, namely by changing the dynamic translation software (e.g., to avoid certain instruction combinations) – a very powerful approach. Still, isolating the hardware bug consumed three weeks of a highly skilled engineer's time, and the bug could not be corrected in time for first tapeout.

We have since developed a methodology which we call *cosimulation*[2], and which successfully addresses this drawback by automating the task of isolating hardware bugs.

The basic idea behind cosimulation is rather simple. Conceptually, our tool runs two separate simulations of the same application, one running on the actual hardware (Verilog) design, and one running on a "known-good" reference model. Periodically (say every million instructions) the tool compares the state (registers, memory, I/O) of the two simulations. If the state matches, both simulations are

[1] This type of bug is sometimes jokingly referred to as a "Heisen-bug" — the more closely you observe it, the more elusive it becomes.

[2] We are aware that some authors and companies use the term cosimulation to refer to *comodeling*, i.e., the mixing of simulation models written in different languages.

checkpointed, and the simulation continues. Otherwise, the tool can isolate the root cause of the discrepancy by performing a binary search over the time interval since the last successful checkpoint, eventually narrowing the window to one or a few processor cycles – all without human assistance.

This cosimulation tool has proven extremely useful during our latest processor design. Used concurrently with traditional validation methods, it has found many logic bugs, and shown up coverage holes in the conventional test suite. We were able to boot several operating systems on the emulator, and run various application programs. All this was achieved before tapeout, and has contributed to a very low bug rate on first silicon.

Referring back to the hardware bug mentioned above which took three weeks' work to isolate on a traditional emulation system, our cosimulation tool would have isolated that bug within an hour, without human assistance.

In the remainder of this paper, we give implementation details of our cosimulation tool and discuss our experience with the system.

2. Related Work

The use of reference models in verification is not new. They have been used to generate test cases, or to verify the final result of Verilog simulations [Bay02, Jah01]. An approach that resembles ours is described in [Gat97], whereby a reference model is simulated in lock-step alongside the register-transfer-level design, and processor state is compared on an instruction-by-instruction basis.

Verifying final simulation results only is valuable in detecting whether there is a bug in the design, yet does not help in the most painful stage of debugging, namely isolating exactly *where* the bug occurred. This is especially problematic when test runs span many cycles, as real-life applications typically do.

On the other hand, comparing state on every cycle or instruction is not practical on a hardware emulator, as it drastically slows down execution and so negates the main advantage of using an emulator.

Several approaches described in the literature rely on *cycle-accurate* reference models; constructing such a model is typically a very time-consuming undertaking. Moreover, maintenance of the reference model can become a nightmare, as many minute changes in the design have to be mirrored.

We address these issues by only comparing state at large time intervals; this allows the emulator to run at maximum possible speed, only slowing down when isolating the source of a failure. By narrowing the window containing the fault to a few processor cycles, we have automated one of the the most tedious and time-consuming aspects of debugging. Moreover, by relying on a fairly high-level reference model (at the architectural level), we avoid the cost typically associated with cycle-accurate models.

3. Cosimulation Overview

Figure 1 shows a block diagram of the cosimulation tool. The *test system* consists of the hardware emulator executing the Verilog hardware design, and some amount of host software (running on a workstation attached to the emulator) which is responsible for interacting with the emulator.

The *reference system* is an architectural (*instruction-level*), "known-good" simulator of the processor. (In fact, Transmeta's dynamic translation software was developed on the reference simulator, greatly increasing confidence in the correctness of the reference simulator.) Both test and reference systems support checkpointing, so that we can perform deterministic *reverse execution* by resuming execution from earlier checkpoints.

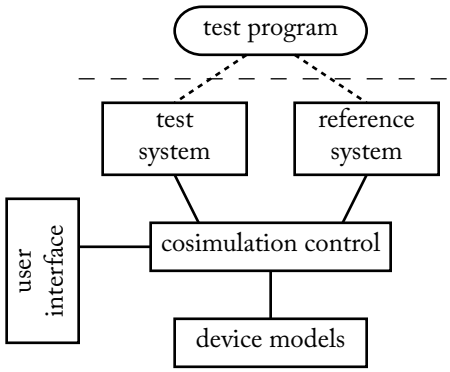

Figure 1. Cosimulation Framework

Both systems execute the same *test program*, typically a real program such as an operating system and/or an application.

Since we are targeting a PC environment, we use a set of *device models*, written in C, that simulate that environment – e.g., floppy disks, harddrives or graphics cards.

Cosimulation proceeds by advancing both test and reference system by some amount of time, e.g. N instructions, after which the state of both systems is compared. If the

state matches, the tool takes checkpoints of both systems and proceeds. Otherwise, it is clear that at least one bug was exposed during the last N instructions, and the tool can now isolate the failure by performing binary search over the interval of the last N instructions.

4. Implementation

The hardware emulator we use, a Mentor VStation 15M model, offers several features that aid implementation of our cosimulation tool. First, it is possible to checkpoint the complete state of the emulator to a file, and restore state from that file later.

Second, *host software* running on a workstation attached to the hardware emulator can control and examine the test system via *transactors*, fragments of Verilog code that are compiled onto the emulator together with the test system. The transactors provide a simple transaction-based interface to the host software, and can control the test system's clock, as well as access signals internal to the test system via convenient cross-scope references.

Third, the Mentor emulator provides "100% visibility", meaning that any and all signals in the test system are always visible to tools, so they can be readily examined after our tool has pinpointed the point where the error occurs.

We have implemented a simple transactor that contains and exposes to the host software

- mechanisms to read and write emulator state elements such as the test system's register file or the data cache, and
- breakpoint logic that stops the clocks to the test system when a given cycle is reached or certain events take place such as an access to main memory.

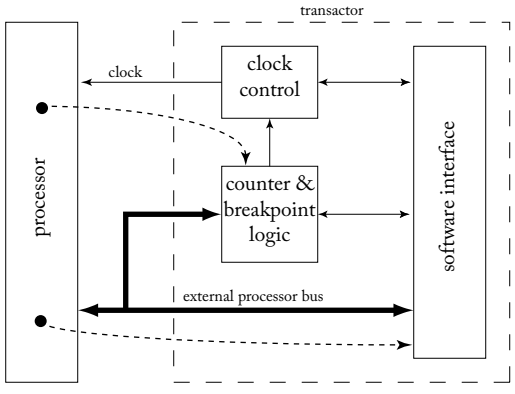

Figure 2. Transactor and Software Interface

Figure 2 shows a block diagram of this setup; note how the transactor logic has access to both the external signals from the processor (shown as thick solid lines) as well as internal signals (dashed lines), typically made available through cross-scope references (i.e., without requiring modification of the Verilog for the processor core).

We use an architectural-level simulator as the reference model; in fact the same simulator has been used to develop and test Transmeta's dynamic translation software; in a sense this simulator is the specification for the processor. (In the next section, we discuss some of the implications of not using a cycle-accurate model as the reference.)

The user-level interface to the cosimulation tool is the Gnu source-level debugger, *gdb*, with some cosimulation-specific commands added.

5. Challenges, Choice of Reference

We have found it desirable to use an *architectural* (or *instruction-level*) simulator as the reference model, as opposed to a cycle-accurate model of the hardware design which might explicitly model all pipeline stages. Reasons to use this approach are:

- An architectural simulator is typically faster, making it suitable for use as a software development platform.
- Being less detailed, and less tied to the actual hardware implementation, an architectural model usually requires less development effort and coordination to keep in sync with the hardware design.
- For the same reason, an architectural model can be reused with few or no modifications, even as details of the hardware implementation change over time.
- The main advantage of a cycle-accurate simulator, namely simpler state comparison, is only available when the match between test and reference is complete down to the smallest detail — something that in our experience is extremely hard to achieve.

On the other hand, the different levels of abstraction between test and reference system cause some complications for cosimulation. Among those issues are

- need for a common time base,
- state extraction and comparison,
- device models and I/O,
- reference model limitations,
- handling of interrupts.

Below we describe these issues and our solutions; overall we have found that the complications incurred by not using a cycle-accurate model are well outweighed by the advantages outlined earlier.

5.1. Common Time Base

Cosimulation obviously requires a definition of "time" common to both test and reference system. Not using a perfectly cycle-accurate reference model rules out the most obvious choice, namely number of cycles executed. Instead, we have instrumented the hardware design so our transactor can count the number of instructions successfully retired, a measure that is natural to the architectural model. This instrumentation was fairly simple, and required little time from the hardware designers.

5.2. State Extraction

Extracting state from a Verilog model of a processor is non-trivial. For instance, at a given cycle, the current value of a simple integer register might not found be in the register file, but instead in the processor's bypass logic. Likewise, the most up-to-date value of a given memory location may be in any of many places in the processor — e.g., the various caches or queues usually part of a modern memory hierarchy. Extracting from the hardware implementation an architectural view of the processor state can therefore be a very involved process, requiring the instrumentation of many functional blocks in the design.

To simplify initial development of the cosimulation tool, we have chosen an approach whereby we *quiesce* the processor before extracting state. Specifically, whenever we wish to extract state, we force a high-priority non-maskable interrupt into the processor; the (software) handler for that interrupt drains all memory queues and feeds a stream of no-ops to the core until all state is settled. This greatly reduces the number of places where state can be found. For example, all values in the integer register file will be current (eliminating the need to extract the state of the bypass network), and memory values exist either in the caches or in main memory.

Since this state extraction is *destructive* (i.e., it changes the state of the processor), we checkpoint the processor before extracting state, and restore from that checkpoint after the state extraction is complete. That way, execution can continue after the state extraction as though nothing had happened. This approach does have certain drawbacks.

- Extra time is spent on taking a checkpoint and restarting from it after the state extraction.
- Our specific implementation for quiescing relies on a high-priority non-maskable interrupt in the hardware and a software interrupt handler in the test program. If either is unavailable, one could augment the test system to directly support quiescing.

It may appear that the state extraction process described here might *hide* latent bugs — for example, the action of quiescing the processor at a given point in time might accidentally "fix up" an already incorrect in-flight operation, so that the state extracted after quiescing does not show the bug. However, sooner or later regular execution will reach a point where the buggy value is committed to a part of the system not perturbed by the act of quiescing.

In other words, at worst the destructive state extraction process could slightly widen the time window within which the bug may be found. Overall, this approach to state extraction has worked very well for us in practice.

5.3. Device Models and I/O

Since we are modelling an entire PC system, we have to verify that both test and reference systems interact with the devices in an "equivalent" manner. This can be difficult to do, especially if the effect of some processor-device interactions depends on exact timing details (which are hard to reproduce in the reference system).

Instead of providing both the test and reference system with their own copy of the device models, we have chosen to use only a single copy of the device models. When the test system advances time, the cosimulation control module *logs* all processor/device interactions. When the reference system is brought to the same point in time, the cosimulation control matches all processor requests against the log, and *replays* any device responses from the log, without involving the actual device models themselves.

One additional benefit of this approach is that device models need only move forward in time, never back, meaning they do not need to support checkpoint and restart functions. With some effort, this may also allow us to use actual (physical) devices at some point, in an ICE (In-Circuit Emulation) type setup.

5.4. Reference Model Limitations

Unless the reference model is extremely accurate and detailed, there will always exist operations where the ref-

erence cannot exactly match the test system. This is fairly noticeable in our case, where the reference system is an instruction-level simulator, without any knowledge of the processor's pipeline, and even without accurate models of caches.

There are several ways of dealing with this problem that do not involve extending the functionality of the reference model:

- Simply prohibit certain operations in the test program being run: in an early development stage of our tool, we replaced any access to the cycle counter by an access to a continuously incrementing software counter. This was particularly easy given Transmeta's dynamic translation approach.
- Log the operation in the test system, and force the same result in the reference: eventually, we instrumented the test system's Verilog so that the transactor could log accesses to resources (such as the cycle counter) that are not properly modeled on the reference system. These logged values are then *forced* into the reference simulation when it is brought up to the same time as the test system. While this approach does lose some test coverage, it also eliminates any perturbation on the test system as described above. Some test coverage can be regained by performing sanity checks on the logged values (e.g. ensuring that the cycle counter increases monotonically).

5.5. Interrupts

Interrupt handling is another instance of where it is hard for a reference simulation to match the test system, without being overly detailed. For example, there is typically a latency of several cycles between the assertion of an external interrupt signal to a processor, and the time when the processor reacts to that interrupt. The number of instructions retired during that interval will vary depending on the exact state of the processor pipeline – e.g., which pipeline stages have bubbles in them, if any, which in turn can depend on the exact state of, say, the branch prediction unit, any prefetch queues, etc. Again, exactly matching all these implementation details in a reference simulator would require a prohibitive amount of work.

Our solution to this problem is to instrument the test system such that the transactor can record exactly when (in terms of the common time base, i.e., number of instructions retired) the test system responds to the interrupt (i.e., dispatches to the interrupt handler). We then delay presenting the interrupt request to the reference system until the time recorded thusly.

Once again, some test coverage is lost — e.g., if the test system never took any interrupts due to a bug, the cosimulation tool would not notice. Some of this test coverage can be regained by performing sanity checks on the interrupt behavior of the test system.

6. Performance

Several components determine the overall speed of the cosimulation system: the "raw" speed of the hardware emulator and the reference simulator, and the cost and frequency of interactions (through the transactor) between the test system and host software. To reduce the latter, we used various approaches.

- For accesses to resources modeled in host software (e.g., main memory in an early incarnation of our tool), have the transactor batch data transfers as much as possible. In the case of a memory read access by the processor, instead of stopping the emulator each time one "strobe" of data (e.g., 64 bits) is to be transferred, we let software deliver all data in a single interaction with the emulator, and rely on the transactor to trickle the data to the processor as required.
- Model main memory entirely in the emulator (possible for most modern emulators). This complicates state extraction a bit and introduces costs of its own, since every time memory needs to be examined, it needs to be uploaded from the emulator. We added per-page dirty bits to indicate to the transactor which pages need uploading since the last extraction.
- When logging events that are to be replayed to the reference system later (as discussed in Section 5.4), have the transactor buffer up many events and deliver them to host software in bulk, at the next time an interaction is required for other reasons (or when the log buffer fills up).
- Increase the *stride*, i.e., the number of instructions between state comparisons. This speeds up forward execution while only slowing down fault isolation (because of the larger time interval within which the fault must be found), which hopefully is the exception rather than the rule once the test system is mostly debugged.

Overall, we have been able to attain cosimulation speeds within a factor of two of the emulator's native speed; this includes the time to run the reference model.

7. Sample Session

In the early phases of debugging our hardware design, we found a bug that caused stores to be dropped under certain circumstances, while booting Windows98. The test suite, still being built up, did not find this bug. A cosimulation session tracking down this bug would look roughly as follows; operator input is shown in boldface:

```
(gdb)select win98-boot emu-cosim

cosim:0(gdb) run
(some amount of time passes)
DIVERGED, simulation stopped
at time 361_824
get_byte () at decompress.c:335
335 byte = *input_data++;

cosim:361_824(gdb) why
Memory diverged at 0xff0106a0;
test system:00000000,
reference:  0000005a

cosim:361_824(gdb) before
Simulation stopped before divergence,
state matches.

cosim:361_821(gdb) wave
Generating waveform, please wait.
```

The session shows that by cycle 361_824, the test system and reference differ at memory address 0xff0106a0, while three cycles earlier, both simulations still match. At this point, it is possible to request from the reference simulator a trace of the last few instructions executed, showing what *should* have happened on the test system. Additionally, our tool lets the user examine the state of various functional blocks in the test system, e.g., the contents of the processor's write queue. The last command in the session generates a waveform including all signals in the processor, covering a time window around the time of divergence. Armed with all this information, it is usually a simple matter for the designer to spot the problem.

In practice, most of the bugs discovered by the cosimulation tool were analyzed by the hardware designers within an hour of being reported.

8. Experience & Discussion

Our cosimulation tool became operational several months before tapeout. We used it concurrently with more traditional validation techniques, while the hardware design was still in flux and the test suites were still being built up. This was possible because of the relatively painless compilation process for the Mentor emulator. Not counting the (preexisting) reference simulator and device models, total development time was approximately nine man-months; a good deal of that time was spent debugging transactor-related Mentor software which at the time of the project was still relatively immature.

In the months leading up to tapeout, the tool found many hardware bugs, and uncovered several coverage holes in the test suites. We have booted several x86 operating systems on the emulator (DOS, Windows 95, Windows 98, OS/2 and Linux), and have run a few application programs (including a game, QuakeII) as well.

We feel that our cosimulation effort contributed strongly to the fact that first silicon was highly functional; there were hardly any bugs in the parts of the chip that were tested in this way.

A somewhat unexpected result was that a concurrent in-circuit emulation (ICE) project benefited greatly from the cosimulation results. The goal of the ICE effort was to debug system-level issues, e.g. board design, or interface with devices. By nature, one must start an ICE effort early, when one can expect the CPU to still be buggy. Additionally, it is often necessary to run full operating systems and applications in order to test third-party devices. Neither of these conditions facilitates debugging. By using cosimulation to debug the CPU core while running the same applications as in ICE, the latter can concentrate on bugs outside the core. We found that once cosimulation became operational, progress on the ICE project also sped up considerably.

One apparent weakness of cosimulation is that it may not find bugs that only manifest themselves for a short period of time, e.g., an incorrect value in a register that is never used, and overwritten with another value shortly afterwards. Reducing the stride of cosimulation (i.e., checking the state more frequently) would reduce the problem though never completely eliminating it (unless the stride was reduced to one cycle, which would render performance unacceptable). Based on the success of our tool at finding bugs, we conclude that such situations are not very common in real-life applications, so this is not a problem in

practice. Note also that as a side effect of I/O logging (see Section 5.3), *any* incorrect interaction with devices, however ephemeral, will be caught.

We also note that cosimulation simplifies generation and debugging of random tests. Two challenges of generating and using random tests are

- ensuring that they detect *whether* a hardware bug is exposed, and
- should a test indicate a failure, determining exactly *where* in the test the failure has occurred.

The former is usually handled by comparing results of operations against "correct" values; obtaining the correct values and ensuring that all operations are covered is the hard part.

The latter problem is harder to attack; the traditional approach is to try to "simplify" (e.g., shorten) the test, or insert additional checks, until the window of suspect operations is small enough to make analysis tractable. As the act of changing the code may in fact hide the bug, this can be a very time-consuming, frustrating and labor-intensive undertaking.

Cosimulation addresses both points rather neatly, with the caveat that results from operations should be "long-lived", e.g. by computing running checksums, using results as inputs to further operations, or by saving values in memory (as opposed to, say, repeatedly computing values in the same register).

9. Conclusions

We have presented a method, named *cosimulation*, for automatically detecting and isolating hardware bugs. The tool works by periodically comparing the execution of our design on a high-speed hardware emulator against a "known-good" reference model. If a discrepancy is found, the tool can isolate the root cause by performing a binary search over the time interval since the last successful state comparison, eventually narrowing the window containing the fault to a few processor cycles – all without human assistance.

We have shown various techniques to make an architectural model serve as the reference; this greatly reduces the implementation cost of our tool as compared to using cycle-accurate reference models.

The value of automatic fault isolation can be judged by comparison with a previous-generation emulator effort at Transmeta. In that instance, a highly qualified engineer spent three weeks chasing down one (fairly simple) hardware bug that was exposed when attempting to boot an operating system. Cosimulation would have pinpointed that bug (and ones much more subtle than that) in a matter of hours, without any human intervention.

In the months leading up to tapeout, cosimulation found many hardware bugs, and uncovered several coverage holes in the test suites. With the detailed information provided by the tool (a full trace of all signals), and a very narrow window of cycles to examine, the hardware designers were usually able to track down (and often fix) the bug within an hour of the report.

A pleasant surprise was that a concurrent in-circuit emulation (ICE) project benefited greatly from the cosimulation results. By using cosimulation to debug the CPU core, the ICE team could concentrate on bugs outside the core. In fact, once cosimulation became operational, progress on the ICE project sped up considerably.

Acknowledgements

The authors would like to thank the engineers at Mentor, especially Doug Warmke and John Stickley, for their absolutely exceptional support. Special thanks also to the fantastic team at Transmeta, and in particular to Eric Hao for his invaluable help in instrumenting the innards of our test system.

References

[Bay02] "Montego™ Design Philosophy, Methodology, and Verification". Bay Microsystems, 700 Augustine Drive, Suite 298, Santa Clara, CA 95054, July 2002. `http://www.baymicrosystems.com/media/BayMicrosystems_Design_Verification.pdf`

[Gat97] James Gateley, "Verifying a Million-Gate Processor". In EEdesign, October 1997. `http://www.eedesign.com/editorial/1997/coverstory9710.html`

[Jah01] Martin Jahner, "How to verify ADSL chips". Broadband Communications Group. In EEdesign, December 2001. `http://www.eedesign.com/story/OEG20011213S0032`

[Kla00] A. Klaiber, "The Technology Behind Crusoe Processors". Transmeta Corporation, 3990 Freedom Circle, Santa Clara, CA 95054, January 2000. `http://www.transmeta.com/about/press/white_papers.html`